CS422 - Programming Language Design

Elements of Object-Oriented Programming

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

During this and the next lecture we will define a simple object-oriented programming language, which will have many of the important features of the more advanced OO programming languages. Object-oriented concepts can be added on top a variety of languages. We call our simple OO language SKOOL.

As for the languages that we defined previously, the very first step is to *understand* the language to be defined. This lecture we will see several *example programs* in our language and will introduce the *basic concepts* of object-oriented programming.

Note: *The lecture notes below currently describe an older version of SKOOL, called OO-FUN (was defined as an extension of FUN); the SKOOL language that we will define using K will be slightly different (it will modify/extend SIMPLE). However, the OO concepts stay the same.*

Objects and Classes

An object can be abstractly thought of as an *encapsulated state*, which one can interact with via an *interface*. The state maps, as usual, names to values. However, the "names" forming the state of an object are called *fields*. The interface allowing one to read or write the state of an object consists of *methods*, which are nothing but functions acting on that object's state. The process of invoking an object's method is often regarded as *sending that object a message* containing the method name and the actual arguments.

Object-oriented programming is a programming language paradigm which facilitates defining, handling and coordinating objects.

It is often the case that several objects are intended to have the same structure, that is, the same fields and the same interface to the outside world. For example, one can have several stacks or

several queues in a program, each with its own particular state. To facilitate defining several objects sharing the same structure, object-oriented programming languages provide *classes*. Then objects are built as *instances* of classes:

```
class c {
  field a
  method initialize() { a := 5 }
  method m() { a }
}
main
  let x = new c()
  in send x m()
```

A program in OO-FUN consists of a set of classes followed by a `main` expression to evaluate. The `new` language construct creates an instance object of a class. A default convention in our language is that, whenever an instance is created, the special method

`initialize` is immediately invoked; this operation is expected to assign initial values to the fields of the newly created object. The `send` construct sends a message, that is a method invocation request, to an object. The above evaluates to `5`.

The following class defines two fields, `i` and `j`; its objects will preserve the invariant `i + j = 0`:

```
class c {
  field i
  field j
  method initialize(x) {
    i := x ;
    j := 0 - x
  }
  method add(d) {
    i := i + d ;
    j := j - d
  }
  method getstate() {
    [i, j]
  }
}
```

Let us now consider the following main expression which creates an object `o` and then sends it the message `add(3)`:

```
main
 let a = 0 and b = 0 and o = new c(5)
 in {
      a := send o getstate() ;
      send o add(3) ;
      b := send o getstate() ;
      [a, b]
    }
```

This will evaluate to the list
`[[int(5),int(-5)],[int(8),int(-8)]]`.

Self References

While evaluating their methods, objects can send messages to themselves in order to invoke other methods. For example, the following evaluates to 13:

```
class c {
  method initialize() { 1 }
  method m1() {
    send self m2()
  }
  method m2() { 13 }
}
main
  let o = new c()
  in send o m1()
```

Self References and Recursion

Recursion can be very elegantly supported by the OO paradigm, because it can be very easily and intuitively explained and handled via message passing:

```
class oddeven {
  method initialize() { 1 }
  method even(n) {
    if n eq 0 then 1 else send self odd(n - 1)
  }
  method odd(n) {
    if n eq 0 then 0 else send self even(n - 1)
  }
}
main
  let o = new oddeven()
  in send o odd(17)
```

Therefore, in some sense, the above is equivalent to a `let rec`.

Dynamic Method Dispatch

One of the most pleasant features of OO programming is the capability of objects to potentially send any messages to each other. Let us consider the following class:

```
class node {
  field left
  field right
  method initialize(l,r) {
    left := l ;
    right := r
  }
  method sum() {
    (send left sum()) + (send right sum())
  }
}
```

Without any apriori knowledge regarding its fields, an object of the class above assumes that both `left` and `right` will be objects providing a method `sum()` in their interface. They can be instances

of node, a subclass of it (subclasses will be discussed in the sequel), or of any other class providing a method sum(). For example, they can be instances of the following class:

```
class leaf {
  field value
  method initialize(v) {
    value := v
  }
  method sum() { value }
}
```

Thus, if an object "knows" that another object is expected to have a certain method as part of its interface, then the former can just send a message invoking that method of the second object. The following will therefore be evaluated to 12:

```
main
  let o = new node(new node(new leaf(3), new leaf(4)), new leaf(5))
  in send o sum()
```

This dynamic style of method invocation is called *dynamic method dispatch*, and it turns out to be of crucial importance in the context of *subclass polymorphism*, which will be explained later.

Most OO programming languages, including Java, use dynamic method dispatch, which is what we will also consider for our language. However, there are languages using *static method dispatch*, such as C++, which have the advantage that allow more efficient implementations. Why? However, due to the practical and methodological importance of dynamic method dispatching, languages like C++ provide so called *virtual* methods, which are dynamically dispatched.

Inheritance

There are many situations in which one would like to define a new class by just slightly modifying an already existing class: adding new fields and new methods to it, or changing the behavior of some existing methods. We say that the new class *inherits*, or *extends*, or is a *subclass* of the old one.

Consider, e.g., the following class defining two-dimensional points

```
class point {
  field x
  field y
  method initialize(initx, inity) {
    x := initx ;
    y := inity
  }
  method move(dx, dy) {
    x := x + dx ;
    y := y + dy
  }
  method getLocation() {
    [x, y]
  }
}
```

One may want to extend it by defining colored points. So one would

like to only define a new field, say `color`, together with methods to read and write it, preserving the already defined behavior of points:

```
class colorpoint inherits point {
  field color
  method setColor(c) {
    color := c
  }
  method getColor() {
    color
  }
}
```

Let us now consider the following scenario, in which a point and a colored point are created:

```
main
 let p = new point(3,4) and cp = new colorpoint(10,20)
 in {
      send p move(3,4) ;
      send cp setColor(87) ;
      send cp move(10, 20) ;
      [send p getLocation(),
       send cp getLocation(),
       send cp getColor()]
    }
```

This should return `[[int(6),int(8)],[int(20),int(40)],int(87)]`.

Let us now consider a situation, in which a subclass redeclares one of its superclass fields:

```
class c1 {
  field x
  field y
  method initialize() { 1 }
  method setx1(v) { x := v }
  method sety1(v) { y := v }
  method getx1() { x }
  method gety1() { y }
}
class c2 inherits c1 {
  field y
  method sety2(v) { y := v }
  method getx2() { x }
  method gety2() { y }
}
```

A subclass' fields can *shadow* some of superclass' fields. Then what is the field accessing rule? It is in fact quite simple and intuitive:

- Any object instance of the subclass will search for its fields first within the subclass definition and then within the superclass definition;
- However, a method invocation will search for the names that occur free in its definition starting with the fields occurring in its class component of the current object instance! Thus, a method always knows statically where its free names are located. That is because we assume *static scoping* in our language.

Thus, the following will evaluate to the list `[int(101),int(102),int(101),int(999)]`:

```
main
  let o2 = new c2() in {
    send o2 setx1(101) ;
    send o2 sety1(102) ;
    send o2 sety2(999) ;
    [send o2 getx1(), send o2 gety1(),
     send o2 getx2(), send o2 gety2()]
  }
```

Our language will be *single inheritance*, that is, its classes are allowed to inherit only one class. Many OO programming languages today are single inheritance. While *multiple inheritance* may look like an obvious extension, it is typically problematic in practice (C++ supports it).

Overriding

If a subclass defines a method which has the same name as a method of its superclass (or some ancestor superclass), we say that the new method *overrides* the old one.

The behavior of overridden methods depends on the dispatch style, dynamic or static. Consider for example the following program:

```
class c1 {
  method initialize() { 1 }
  method m1() { 1 }
  method m2() {
    send self m1()
  }
}
class c2 inherits c1 {
  method m1() { 2 }
}
main
  let o1 = new c1() and o2 = new c2()
  in [send o1 m1(), send o2 m1(), send o2 m2()]
```

Which `m1()` is meant in the expression body of `m2()` within an instance object of `c2`?

Under *dynamic method dispatch* it refers to the `m1()` defined in `c2`, while under *static method dispatch* to the `m1()` defined in `c1`. So the above evaluates to `[1,2,2]` in the first case and to `[1,2,1]` in the second.

Exercise 1 *What list does the following program evaluate to under dynamic dispatch?*

```
class c1 {
  method initialize() { 1 }
  method m1() { 1 }
  method m2() { 100 }
  method m3() {
    send self m2()
  }
}
class c2 inherits c1 {
  method m2() { 2 }
}
main
  let o1 = new c1() and o2 = new c2()
  in [send o1 m1(), send o1 m2(), send o1 m3(),
      send o2 m1(), send o2 m2(), send o2 m3()]
```

Invoking Superclass Methods

There are situations when one wants to refer to a superclass method, despite the fact that it has been overridden. Consider again the extension of points with colored points:

```
class point {
  field x
  field y
  method initialize(initx, inity) {
    x := initx ;
    y := inity
  }
  method move(dx, dy) {
    x := x + dx ;
    y := y + dy
  }
  method getLocation() {
    [x, y]
  }
}
```

The following shows one possible definition of its extension `colorpoint` class, which has its own `initialize` method:

```
class colorpoint inherits point {
  field color
  method initialize(initx,inity,initcolor) {
    x := initx ;
    y := inity ;
    color := initcolor
  }
  method setColor(c) {
    color := c
  }
  method getColor() { color }
}
```

The problem with the above is that it *repeats* the initialization code of the superclass, which may be inconvenient in large examples. In order to solve this problem, we can use a new language construct,

`super`, which calls the corresponding method of the superclass:

```
method initialize(initx,inity,initcolor) {
  super initialize(initx, inity) ;
  color = initcolor
}
```

Exercise 2 *What value does the following program evaluate to?*

```
class c1 {
  method initialize() { 1 }
  method m1() {
    send self m2()
  }
  method m2() { 13 }
}
class c2 inherits c1 {
  method m1() { 22 }
  method m2() { 23 }
  method m3() {
```

```
    super m1()
  }
}
class c3 inherits c2 {
  method m1() { 32 }
  method m2() { 33 }
}
main
  let o3 = new c3()
  in send o3 m3()
```

A More Complex Example

```
class a {
  field i
  field j
  method initialize() 1
  method setup() {
    i := 15 ;
    j := 20 ;
    50
  }
  method f() send self g()
  method g() i + j
}
```

```
class b inherits a {
  field j
  field k
  method setup() {
    j := 100 ;
    k := 200 ;
    super setup() ;
    send self h()
  }
  method g() [i :: j :: k]
  method h() super g()
}
```

```
class c inherits b {
  method g() super h()
  method h() k + j
}
```

```
main
  let p o = let u = send o setup()
            in [u :: send o g() :: send o f()]
  in [p(new a()) :: p(new b()) :: p(new c())]
```

Creating Objects in Methods

Since objects are regarded as any other values in the language, one can also create objects within methods and pass or return them. For example, the method `m1` below creates and returns an object of class `g`:

```
class i {
  field value
  method initialize(v) {
    value := v
  }
  method m1() {
    new g(value + 9)
  }
}
```

The class `g` can, for example, be the following:

```
class g {
  field a
  method initialize(t) {
    a := t
  }
  method get() { a }
}
```

Then if one creates an object of `i` and sends it the message `m1()`, then one gets an object of class `g` to which one can send the message `get()`:

```
main
  send (send (new i(1)) m1()) get()
```

The above therefore evaluates to `10`.

Exercise 3 *What value does the following* OO-FUN *program evaluate to?*

```
class i {
  field f
  method initialize(v) {
    f := v
  }
  method get() { f }
}
class g {
  field o
  method initialize(obj) {
    o := obj
  }
  method do() {
    let f = 9
    in  send o get()
```

```
  }
}
main
let t = new i(1)
in let y = new g(t)
   in send y do()
```